JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **11**, 117–142 (1999)

Research

Variable-precision Reaching Definitions Analysis

P. TONELLA[1]*, G. ANTONIOL[1], R. FIUTEM[1] and E. MERLO[2]

[1] *IRST—Istituto per la Ricerca Scientifica e Tecnologica, I-38050 Povo, Trento, Italy*
[2] *Department of Electrical and Computer Engineering, Ecole Polytechnique, C.P. 6079, Succ. Centre Ville, Montréal, Quebec, Canada*

SUMMARY

Ascertaining the reaching definitions from the source code can give views of the linkages in that source code. These views can aid source code analyses, such as impact analysis and program slicing, and can assist in the reverse engineering and re-engineering of large legacy systems. Maintainers like to do such activities interactively and value fast responses from program analysis tools. Therefore the control of the trade-off between accuracy and efficiency should be given to the maintainer. Since some real world programs, especially in languages like C, make much use of pointers, and efficient points-to analysis should be integrated within the computation of the data dependencies during the process of ascertaining the reaching definitions.

This paper proposes three different approaches to the analysis of the reaching definitions based on different levels of precision, reflecting differences in their sensitivity to the calling context and the control flow. The least precise approach produces an overestimate by an average of 41% of data dependencies compared to the approach with the highest degree of precision. The result for the least precise approach is conservative because all detectable data dependencies are included, and is far faster than the more precise approaches. Runs on a test suite show an almost 2000 to 1 reduction in execution time by the least precise approach compared with the most precise approach. The intermediate approach is more than 30 times faster than the most precise approach, and much more precise than the least precise one (an average of 2% extra dependencies compared to the most precise approach). Therefore, while on medium size systems the intermediate approach could be a good compromise, on large systems the least precise approach becomes extremely valuable, being the only one feasible. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: re-engineering; reverse engineering; flow analysis; reaching definitions; program understanding; legacy systems

1. INTRODUCTION

Program maintenance activities are often very difficult because the maintainer is not familiar with the code, and the source has become unreadable as a consequence of successive interventions. In

*Correspondence to: P. Tonella, Istituto per la Ricerca Scientifica e Tecnologica, loc. Panté di Povo, I-38050 Trento, Italy. Email: tonella@irst.itc.it

CCC 1040–550X/99/020117–26$17.50

Received 30 June 1997
Revised 22 December 1998

many cases, comments are rare or inconsistent with the program, and either the documentation has never been produced or it is not up to date.

During maintenance, when it is important to keep all possible linkages under control, a reaching definitions analyser can be used to determine the code fragments that may have defined the value of a variable reaching a given program point. Flow analysis algorithms, which have traditionally been developed in the framework of optimizing compilers and program parallelization and vectorization (Emami, Ghiya and Hendren, 1994; Horwitz, Pfeiffer and Reps, 1989; Horwitz, Reps and Binkley, 1988; Landi and Ryder, 1992), have recently been investigated in the context of software maintenance and program understanding. Their application ranges from slicing to change/impact analysis and many tools that perform architecture recovery (Fiutem *et al.*, 1996a,b; Harris, Reubenstein and Yeh, 1995) and concept extraction (Kozaczynski, Ning and Engberts, 1992) also depend on data dependence computation.

The present work is aimed at comparing the reaching definitions algorithms obtained when flow/context sensitivity is successively varied, and evaluating the trade-off between efficiency and accuracy among the variants. Program analysis tools should run in a reasonable time to be effective during maintenance activities, and therefore the user should be given control of the trade-off between running time and precision of the results.

Several algorithms, with different sensitivity to control flow and calling context, have been developed to obtain accurate and efficient results on reaching definitions. While traditionally flow sensitive analyses received great attention from the research community (Callahan, 1988; Emami, Ghiya and Hendren, 1994; Pande, Landi and Ryder, 1994; Wilson and Lam, 1995), recently flow insensitive analyses have been investigated for their efficiency, and the trade-off between time complexity and accuracy has been evaluated (Steensgaard, 1996a) with regard to the points-to analysis. In Horwitz, Reps, and Binkley (1988) the problem of handling the effects of different calls to the same procedure has been termed the *calling context problem*. In Pande, Landi and Ryder (1994) this has been restated as the problem of restricting the propagation of flow information only along *realizable* interprocedural paths, i.e. from a call node into the called procedure, and back from the return node of the procedure into the same call node. Pande, Landi and Ryder (1994) proposed an algorithm that computes accurate interprocedural reaching definitions in the presence of single level pointers.

Such analysis is flow sensitive, and the calling context is abstracted by means of an *assumed reaching definition*, where the validity of a reaching definition is conditional upon the validity of the assumed reaching definition at the entry of the enclosing procedure. An alternative approach which ensures context sensitivity was proposed by Emami, Ghiya and Hendren (1994), where the explicit construction of the invocation graph provides each procedure with a distinct calling context for each possible call sequence starting from the main procedure. The work described in Wilson and Lam (1995) combines context sensitivity of Emami, Ghiya and Hendren (1994) and efficiency, by summarizing the effects of a procedure by means of a *transfer function*. On the other hand many other approaches, inspired by Weiser (1984), do not account for the calling sequence, and are therefore context insensitive.

As described above, there are many papers related to each analysis variant taken separately but none of them, to our knowledge, compares them with each other with regard to the problem of reaching definitions. The most related works in such a framework are Atkinson and Griswold (1996) and Stocks *et al.* (1998). In Atkinson and Griswold (1996) the importance of controlling the time

complexity by varying context sensitivity was investigated with regard to program slicing. The user is allowed to specify the desired context sensitivity level, resulting in a graph where contexts are merged from a given depth down. In Stocks *et al.* (1998) flow and context sensitivity are considered with reference to the problem of computing pointer-induced aliases. A flow-sensitive context-sensitive algorithm for alias analysis is compared with a flow and context insensitive counterpart by measuring the number of side effects determined by the two variants. The results suggest that the differences in accuracy are substantial, but there are scalability problems for the more accurate algorithm above the 10 000 LOC (lines of code).

The following three variants of the interprocedural reaching definitions analysis are investigated in this paper, differing in the sensitivity to the calling context and to the control flow:

(i) context-insensitive flow-insensitive (CIFI) analysis;
(ii) context-insensitive flow-sensitive (CIFS) analysis;
(iii) context-sensitive flow-sensitive (CSFS) analysis.

Experimental results on a test suite of C programs are presented in this paper, and the trade-off between accuracy and computational resources required is discussed. Analysed programs are public domain applications, mostly chosen among system utilities, and exploit all the complex features of the C language (e.g., preprocessor macros, recursion, pointers, function pointers and structures).

The paper is organized as follows: Section 2 presents the variable precision interprocedural reaching definitions analyses. Section 3 describes the experimental set-up and results. Section 4 is devoted to conclusions.

2. VARIABLE PRECISION REACHING DEFINITIONS

The *reaching definitions* of a variable x at a program point n are the set of control flow graph (CFG) nodes $n_1, \ldots, n_i, \ldots, n_k$ which satisfy the following properties: each n_i defines x, and there exists a path in the CFG from each n_i to n along which there is no redefinition of x, i.e. along which no statement writes the content of x. The *data dependencies* of a variable x at a program point n are the set of reaching definitions of x at n with the additional constraint that n uses x. The data dependencies between n and $n_1, \ldots, n_i, \ldots, n_k$ are also called use definition chains (ud-chains). Reaching definitions can be computed by propagating appropriate flow information inside the CFG of the program until the fix point is reached. Such propagation is flow sensitive, since it follows CFG edges, thus considering only structurally possible execution paths. Each flow information item, a pair $\langle x, n \rangle$, is a representation of the definition of variable x at node n. The rules for the propagation of flow information from each predecessor p of node n to such a node are expressed by the following equations:

$$\mathrm{IN}_n = \bigcup_{p \in \mathrm{pred}(n)} \mathrm{OUT}_p \tag{1}$$

$$\mathrm{OUT}_n = \mathrm{GEN}_n \bigcup (\mathrm{IN}_n - \mathrm{KILL}_n) \tag{2}$$

where IN_n and OUT_n contain incoming and outcoming flow information and are updated during flow propagation, while GEN_n and KILL_n are fixed for every node and contain the flow information that is generated or killed. Each node *generates* a definition for each variable it defines, and *kills*


```
int a;

main()
{
1     a = 1;
2     f(2);
3     printf("%d\n", a);
4     a = 3;
5     printf("%d\n", a);
6     f(4);
7     printf("%d\n", a);
}


f(int x)
{
8     if (x)
9         a += x;
10    return;
}
```

Figure 1. Example of C source code. Function `f` *is invoked in two different contexts (*`main-2` *and* `main-6`*)*

every other incoming definition of the same variable, as expressed by the following equations:

$$\mathrm{GEN}_n = \{\langle x, n\rangle : n \,\mathrm{def}\, x\} \tag{3}$$

$$\mathrm{KILL}_n = \{\langle x, n'\rangle : n \,\mathrm{def}\, x, n' \in N\} \tag{4}$$

A flow-insensitive computation ignores the CFG structure, and collects the definitions from every GEN set.

When interprocedural analysis is taken into account, the reaching definitions that are propagated inside a called procedure and back on return from it depend on the calling context. A *realizable path* in the interprocedural CFG is a path such that whenever a procedure on the path returns, it returns to the call site which invoked it (Pande, Landi and Ryder, 1994). An interprocedural analysis which propagates flow information only along realizable paths is context sensitive in that it returns flow information only to the calling context from which flow information was transferred to the called procedure.

Sensitivity to calling context and control flow influences the accuracy of the reaching definitions computation. Lower sensitivity produces higher efficiency (in terms of time and space) at the expense of precision. With reference to the example code in Figure 1, three levels of sensitivity are presented in Section 2.1: context-sensitive flow-sensitive (CSFS), context-insensitive flow-sensitive (CIFS) and context-insensitive flow-insensitive (CIFI) analyses (see Table 1). They are in decreasing precision order: CSFS analysis is the most accurate, since both the calling context and the control flow are taken into consideration. CIFS analysis does not distinguish among different activations

Table 1. Flow and context sensitivity combinations

	Flow	
Context	Sensitive	Insensitive
Sensitive	CSFS	
Insensitive	CIFS	CIFI

(contexts) of the same procedure. CIFI analysis ignores the flow of control as well. As the calling context of a procedure depends on the control flow, it is not possible to be fully context sensitive without being flow sensitive. For this reason the context-sensitive flow-insensitive (CSFI) variant will not be considered in the following.

2.1. Overview

The source code of Figure 1 is a C program, consisting of a `main` which calls the procedure `f` in two different contexts (Statements 2 and 6). Both `main` and `f` define the value of the global variable `a`. The second definition in `main` (Statement 4) overrides (*kills*) every previous definition, as it holds on every path. On the other hand the definition of `f` (Statement 9) is *non-killing* for Statement 10 and for the calling procedure, since an alternative path does exist along which `a` is not defined (from 8 directly to 10).

A *calling context* is the sequence of procedures and call nodes encountered before entering a called procedure (in Figure 1 procedure `f` is called in two different contexts: `main-2` and `main-6`). It is possible to compute a precise analysis if the calling context is taken into consideration for each single procedure (context-sensitive analysis). If calling contexts are not distinguished, the flow information entering a procedure is cumulated with that coming from all possible contexts. As a result the cumulative flow information exiting from a procedure is propagated back to every calling context and a call node can receive back some information that should be returned only to other call nodes. Therefore, some spurious reaching definitions can be generated by the context-insensitive analysis, due to the propagation of some flow information along unrealizable paths (coming from a context and returning to another context) (Pande, Landi and Ryder, 1994). However, the resulting reaching definitions represent a conservative approximation of the exact outputs, since every realizable path is surely traversed. As flow-sensitive analyses consider all and only the execution paths allowed by the CFG, *strong updates* of the flow information can be performed. This means that when a definition of a variable is encountered at a node, every definition entering that node is *killed* by the current definition and consequently it is not forward propagated to successive nodes. If every path to a successive node crosses the killing node, the killed definitions cannot reach it. When the control flow is ignored, all the definitions inside a procedure have to be collected, and they have to be considered to hold at every node of the procedure. Since there is no notion of precedence among nodes, it is not possible to state that from a certain node on, previous definitions are killed. As a result some spurious definitions are reported, since they are not killed when they could be. Nevertheless, every definition that holds is assured to be recognized.

Figure 2 represents the control flow graphs (CFGs) of the two functions of the example in

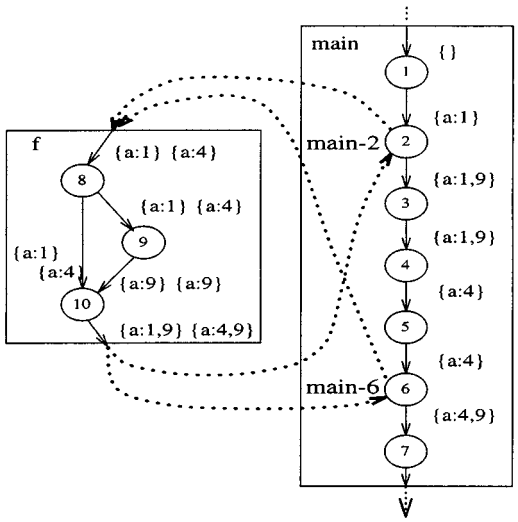

Figure 2. CSFS reaching definitions for variable a. *The list of possible defining nodes follows variable name inside curly brackets. For the two different calling contexts of function* f *flow information is kept separate*

Figure 1. Call and return edges are depicted with dotted lines, and represent the interprocedural links between the functions. Reaching definitions are depicted on each edge, inside curly brackets. They are reported as the name of the variable followed by the list of possible defining nodes. This is a summary notation for the set of pairs $\langle x, n \rangle$. Function f receives its incoming flow information from the two call nodes, numbered 2 and 6. The reaching definitions associated with these two calls are kept separate as shown in Figure 2: the CSFS algorithm performs one propagation for each calling context, and therefore two distinct propagations for f. The first propagation results in the reaching definitions set $\{a : 1, 9\}$, which is returned only to its respective call site, i.e. node 2. The second propagation receives the incoming flow information from node 6, $\{a : 4\}$, and results in $\{a : 4, 9\}$, which is returned to 6 only. The reaching definitions of function f, with no regard to a particular calling context, are computed as the union of the flow information from every calling context.

Figure 3 reports the CIFS results. Flow information entering function f, $\{a : 1, 4\}$, is the union of information from every calling context. After propagation inside f the outcoming reaching definitions are $\{a : 1, 4, 9\}$. This flow information is returned to every call site, even along unrealizable paths, giving rise to spurious definitions. So, for example, the pair $\langle a, 4 \rangle$ is propagated into f by node 6. This same pair is still present at the final node of f, because f does not kill it, and is returned to both call sites 2 and 6. Therefore, an unrealizable path (call from 6, propagation inside f, return to 2) is traversed, thus generating a spurious definition, $\langle a, 4 \rangle$, at node 2.

Figure 4 reports the CIFI results. Definitions of the global variable a are collected from every procedure in a non-killing way, regardless of the calling context and the control flow, by visiting the CFG nodes in an arbitrary order. The result is the set of reaching definitions $\{a : 1, 4, 9\}$ that holds at every node in the scope of the variable (the whole program, in this case).

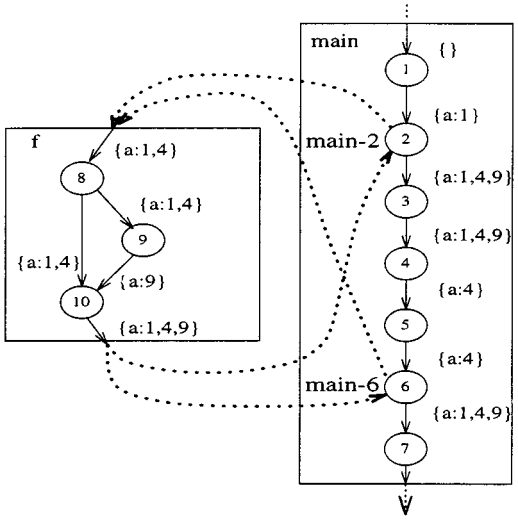

Figure 3. CIFS reaching definitions for variable a. *The list of possible defining nodes follows variable name inside brackets. For the two different calling contexts of function* f *flow information is merged and returned to both call sites*

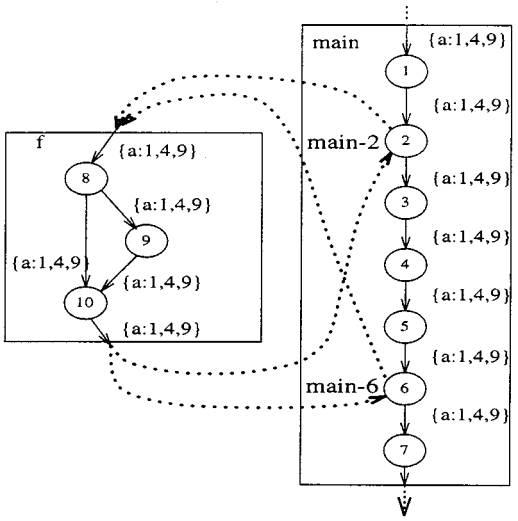

Figure 4. CIFI reaching definitions for variable a. *The list of possible defining nodes follows variable name inside brackets. Calling contexts and control flow are ignored, and definitions are collected in a non-killing way*

Table 2 summarizes the reaching definitions at Instructions 3, 5, 7 on variable `a` for the `main` function of the example in Figure 1. Reaching definitions on `a` at Statement 3 are $\{a : 1, 9\}$ according to the CSFS analysis, i.e. the value of `a` printed at Statement 3 may have been defined by 1 or by 9. At 5 only one definition is active, that of Statement 4. Any control flow along `main` and `f` reaching node 5 makes valid the definition of 4 and kills every previous definition. At Statement 7 the reaching

Table 2. Reaching definitions on variable `a` at nodes 3, 5 and 7 for the example code according to the three variable precision analyses

Statement	CSFS	CIFS	CIFI
3	{1, 9}	{1, 4, 9}	{1, 4, 9}
5	{4}	{4}	{1, 4, 9}
7	{4, 9}	{1, 4, 9}	{1, 4, 9}

definitions are $\{a : 4, 9\}$. The CIFS analysis produces a different result at Statements 3 and 7. As calling contexts are not distinguished, the flow information from `main-2` and `main-6` is merged and, after propagation, is returned to both call nodes (2 and 6). As a result the spurious definition of node 4 is reported at Statement 3, and the definition of 1 at 7. The CIFI analysis computes a set of reaching definitions which holds at every point in `main`. The resulting set $\{a : 1, 4, 9\}$ is an approximate conservative estimate of the definitions computed by the other analyses. It introduces two spurious definitions, $\{a : 1, 9\}$, not reported by the CIFS analysis, at Statement 5: flow sensitive analyses identify the definition at node 4 as a killing definition for the incoming definitions, which are not forward propagated. This is not possible in CIFI analysis.

2.2. Pointers and function pointers

Pointers and function pointers are handled with an approach similar to Steensgaard (1996a,b), and described in Tonella *et al.* (1997): a CIFI points-to analysis is first performed, and the resulting points-to sets are used to determine the locations that are actually defined or used by each statement. This information is necessary for all of the variable precision reaching definitions analyses.

In the presence of calls through function pointers, the points-to set of the involved pointer allows identification of possibly invoked functions, and consequently allows the construction of the call graph, which is necessary for the analyses presented.

A function may define a variable that is neither local nor global through a pointer dereference. Such a variable is *invisible* inside the given function, but still accessible through a pointer. It will be called *invisible–accessible* in the following. To properly collect the definitions of invisible–accessible variables, they are treated as a particular case by the flow-sensitive algorithms (CSFS and CIFS). They do not need special treatment in the CIFI algorithm, because in its first step all definitions are collected from everywhere, even those related to invisible–accessible variables and obtained through pointer dereference. In the CSFS and CIFS algorithms the flow information of the invisible–accessible variables is transferred inside the called function by extending their scope of visibility. In this way, if a called function defines an invisible–accessible variable through pointer dereference, such a definition can be properly added to the current definitions and returned to the function in which such a variable is visible.

Figure 5 shows an example in which variable `x` is invisible inside `f`, but is still accessible through the dereference of pointer `q`. In fact the points-to analysis performed in this program suggests that both pointers `p` and `q` may point to variable `x`. Therefore, the assignment at line 6 is a definition of variable `x`, even if it is not directly accessible in `f`. The GEN sets of program nodes are shown in the right column of Figure 5. At line 6 the GEN set includes the definition of the invisible–accessible


```
main()
{
      int x, *p;

1     x = 1;                          GEN = {x:1}
2     p = &x;                         GEN = {p:2}
3     f(p);                           GEN = {}
4     printf("%d\n", x);              GEN = {}
}

f(int* q)
{
5    if (-)                           GEN = {}
6         *q = 2;                     GEN = {x:6}
7    return;                          GEN = {}
}
```

Figure 5. Example of C code, where variable x *is invisible but accessible inside* f

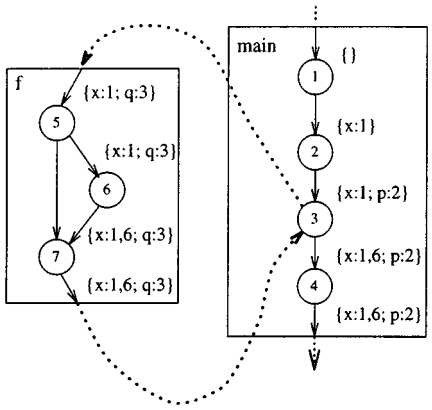

Figure 6. Flow sensitive analysis in the presence of the invisible variable x*. The reaching definitions of* x *are propagated inside* f*, where the variable is reachable through the pointer dereference mechanism*

variable x. Having been recognized as invisible–accessible, x has to be treated with special care during flow-sensitive propagations, and its flow information has to be transferred to the called function, where access to it is possible through pointer dereference.

Figure 6 depicts the reaching definitions for the example in Figure 5. Definition {x:1} is propagated at entry of function f, since x is an invisible–accessible variable. In turn, function f modifies the reaching definitions of x into {x:1,6}, which are returned to call node 3. Definition {p:2} is not transferred into f as p is local to main and not invisible–accessible, it being invisible but not accessible. Finally, it can be noted that a definition is generated for the formal parameter q.

It contains a reference to call node 3, since the expression used as the actual parameter in 3 defines the initial value of the formal parameter.

A more formal description of the presented algorithms is given in Appendix A, where computational complexity issues are also discussed.

3. EXPERIMENTAL RESULTS

The algorithms described above have been implemented in FLANT (flow analysis tool), which is a module of the program understanding and re-engineering environment CANTO (code and architecture analysis tool) (Antoniol *et al.* 1995).

The organization of FLANT is shown in Figure 7. The source code is parsed by a front-end module, which translates it into an intermediate language. The preprocessor macro values are specified in the Configuration file. The information specified in the intermediate language Abstract Syntax Tree (AST) is both structural and semantic. Structural information specifies the organization of the program into procedures, and the arrangement of procedure nodes in a syntax tree. Semantic information includes the GEN and KILL sets of each node, and the points-to relations introduced by each node. The front end can also access a database of function models, used during the translation from the original programming language into the intermediate language. A function model, similar to that proposed in Weiser (1984), is used whenever a statement invokes a function the source code of which is not part of the system under analysis. It can either be a library function or a function belonging to a module that is not considered in the current analysis. A function model specifies the definitions introduced by the modelled function and visible outside it. Such definitions may refer both to global variables and to variables that are local to the calling function, but accessible through pointer dereference in the modelled function. The availability of these models allows one to properly determine the GEN and KILL sets of call nodes that invoke a function in which the analysis cannot descend, as its code, and therefore its AST, is not given.

The format of the AST is highly independent from the programming language of the system under analysis. Given a front end which translates the original language into the intermediate language, it is possible to use the same analyser for different imperative languages. In CANTO, the intermediate representation for C code programs is generated by a Refine† based front-end program. By decorating the AST with additional control flow information (e.g. the target node of each `goto` statement), the reaching definitions analyses can be performed directly on the AST, with no need to generate the CFG from the AST. In fact, FLANT performs all of its analysis directly on the AST.

The intermediate language AST is first analysed to extract the points-to relations that hold between program locations. The points-to analysis used for this purpose is described in Tonella *et al.* (1997). The resulting points-to relations are accessed by an integration unit that transforms the AST into an augmented AST+. The AST+ differs from the AST in that every arbitrary chain of pointer dereferences is replaced by the set of memory locations reachable through such chain. In addition, when a call node exploits function pointers to specify the called function, it is transformed into a polymorphic call node, and, as concerns the invoked functions, the dereference chain of the function pointer is replaced by the set of functions reachable through such chain. The AST+ code representation also solves name conflicts: variables in the AST+ are given unique identifiers. The

†Refine and Refine/C are trademarks of Reasoning Systems Inc.

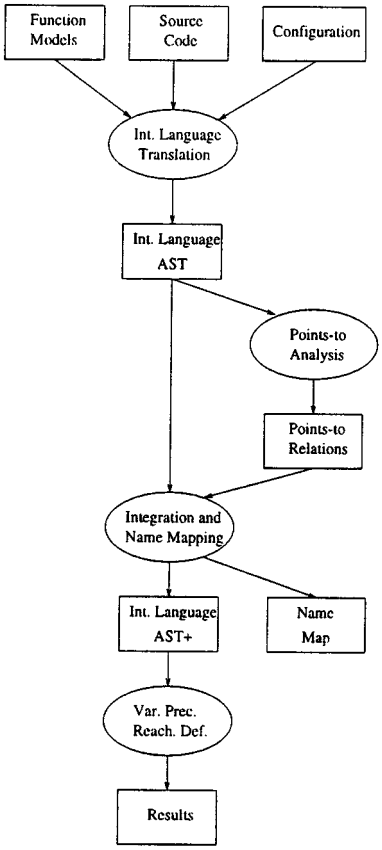

Figure 7. Organization of FLANT

table of correspondence that allows restoration of original names, when needed, is in the Name Map file. At this point the intermediate language representation of the program is in a format suitable for the variable precision reaching definitions analyses.

3.1. Test suite

The test suite contains real public domain applications, many of which are common system utilities. They are representative of the expressive capabilities of the C language, as, e.g. pointers (and pointers to functions), structures, recursion and dynamic allocation.

Table 3 reports some features of the programs used to test the three variants of the reaching definitions computation. Their size ranges from 33 lines of code (LOC) to 16 474 LOC, while the number of procedures is between 1 and 201. The three size measures, LOC, uncommented LOC (ULOC) and number of AST nodes, have similar plots, with some exceptions like `wc` and `shorten`, which have respectively less LOC than `whoami` and `gdbm`, but more nodes. It is

Table 3. Some features of the programs in the test suite: lines of code (LOC), uncommented LOC (ULOC), AST nodes, procedures and calling contexts

Program	Version	LOC	ULOC	Maximum nodes per procedure	AST nodes	Procedures	Contexts
`sizeof`		33	26	14	14	1	1
`d2j`		92	71	24	55	4	3
`div`		161	109	78	150	3	10
`crc`		428	182	39	102	6	15
`xref`		492	336	118	342	7	25
`fft`		621	532	212	438	8	15
`yes`		1 796	942	249	416	10	11
`wc`		1 916	1 110	250	671	14	28
`whoami`		2 004	1 049	249	445	10	12
`shorten`	1.23	2 525	1 896	1 381	3 012	50	411
`gdbm`	1.7.3	7 025	3 665	583	2 357	54	872
`gzip`	1.2.4	8 163	5 215	439	5 358	98	10 134
`grep`	2.0	12 935	8 154	2 632	11 204	130	27 783
`find`	4.1	16 474	10 684	2 632	12 043	201	538 693

expected that execution times of reaching definitions analyses are correlated with the number of CFG (AST for FLANT) nodes of a program, so that `whoami` and `gdbm` analysis should reveal a computation faster than `wc`'s and `shorten`'s.

The number of nodes in each procedure plays an important role in reaching definitions algorithms complexity. The presented flow-sensitive algorithms perform a fix point inside a procedure every time such a procedure needs to be considered. The fix point complexity depends on the maximum number of nodes internal to procedures. One could think that the maximum nodes number is a constant among programs. On the contrary, Figure 8 shows relevant variations from program to program, as, e.g. for `shorten`. The two programs with the highest value, `grep` and `find`, use the same module for regular expressions, `regex.c`, which contains the function responsible for the peak, `regex_compile`. The test suite is too small to state that the maximum number of nodes is an increasing function of program size. However, on the two programs for which the maximum is the highest, the performance of the analyses is expected to get worse, not only due to the higher number of procedures, but also because of the increased maximum nodes number.

The number of the calling contexts in each program makes the complexity of the context-sensitive approach grow. The high number of contexts in the `find` program is due to the heavy use of function pointers, which are polymorphically resolved by our points-to algorithm (Tonella *et al.* 1997). Figure 9 gives the number of calling contexts as a function of the number of procedures in the program. The plotted value is the number of different sequences of call nodes and called functions starting from the `main` and excluding recursive invocations. Theory suggests that a worst case exponential relation between the two is the main reason for the higher complexity of the presented context-sensitive analysis, with respect to the context-insensitive ones. The test suite is too small to draw definitive conclusions, but the data indicate that such exponential dependence is not only theoretically possible, but has also some empirical evidence, since the plot in Figure 9 resembles an exponential curve. For this reason the presented CSFS analysis is expected not to scale on the largest programs in the test suite.

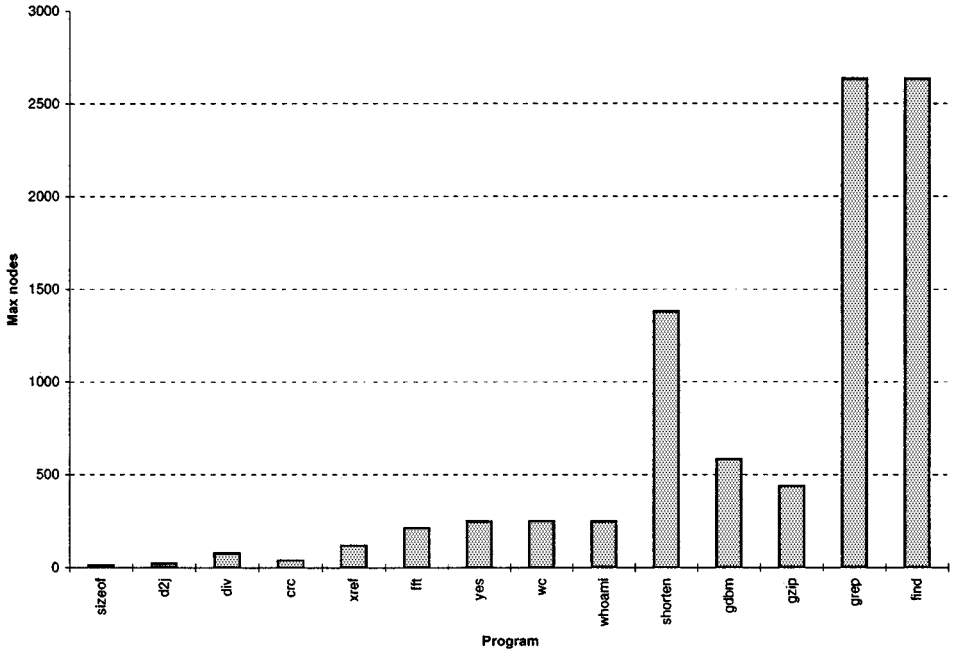

Figure 8. Maximum number of nodes per procedure

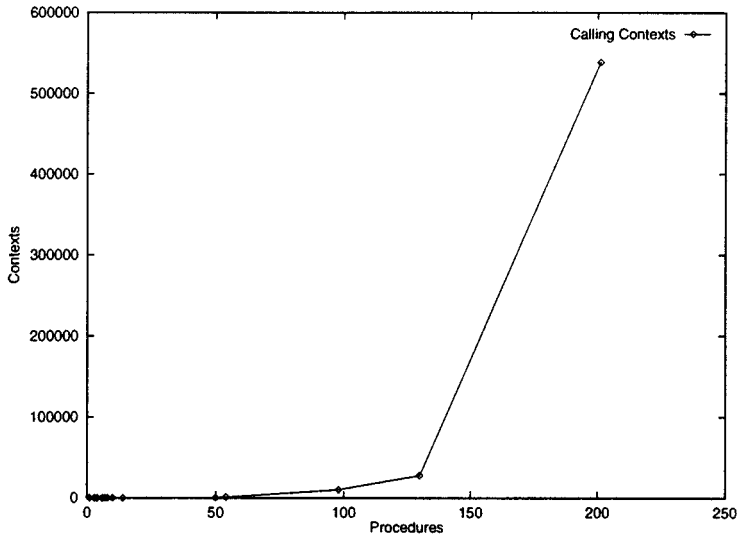

Figure 9. Number of calling contexts as a function of the number of procedures

Table 4. Reaching definitions computation times (s) and data dependencies count for the three variants of the analysis. Times higher than five hours of computation are reported in brackets

	Program	LOC	Time (s)			Data dependencies		
			CSFS	CIFS	CIFI	CSFS	CIFS	CIFI
1	`sizeof`	33	0.01	0.01	0.01	10	10	10
2	`d2j`	92	0.2	0.1	0.01	41	41	70
3	`div`	161	2.9	1.2	0.1	86	89	153
4	`crc`	428	3.1	0.4	0.01	68	68	74
5	`xref`	492	256.7	7.0	0.2	499	499	675
6	`fft`	621	1 857.8	8.1	0.2	9 832	9 832	10 935
7	`yes`	1 796	64.3	5.9	0.2	627	671	1 357
8	`wc`	1 916	164.5	19.5	0.3	1 294	1 371	1 685
9	`whoami`	2 004	41.7	6.7	0.2	1 054	1 128	1 292
10	`shorten`	2 525	(5h47m)	893.5	1.5	—	34 170	56 570
11	`gdbm`	7 025	(14h26m)	776.6	1.3	—	9 275	11 037
12	`gzip`	8 163	—	—	4.0	—	—	279 892
13	`grep`	12 935	—	—	8.5	—	—	1 007 430
14	`find`	16 474	—	—	9.8	—	—	2 032 110

3.2. Results

Table 4 reports the results of the reaching definitions computation on the test suite. Execution times are given for the three variants of the analysis, along with (last three columns) the number of data dependencies determined by each variant. The CIFS and the CIFI analyses conservatively retrieve more dependencies than their CSFS counterpart. Times higher than the five-hour limit are in brackets, when available.

The results on execution times are also graphically shown in Figure 10. The presence of a peak corresponding to the `fft` program can be seen for the CSFS analysis. A second peak on `wc` is present for both the CSFS and the CIFS analyses. The CSFS execution time was under the five-hour limit for the first nine programs and consequently the CSFS plot ends on `whoami`. In fact the number of contexts becomes considerably high from the successor of `whoami`, and thus the CSFS analysis becomes excessively expensive. The plot of the CIFS analysis ends on `gdbm`, since the number of procedures in the program rapidly increases after it, and consequently computation time exceeds the five-hour limit. The CIFI analysis could be conducted for all the programs in the test suite, and showed a small increase only on the largest programs.

The CIFI variant gains much over the CSFS analysis, with a high peak on `fft`. The CIFS variant also gains much on `fft`, while CIFI gains much over CIFS analysis on programs `shorten` and `gdbm`, because of the effect of the high number of procedures. In Fact, for these two programs the CSFS analysis could not be conducted within five hours of computation. Average execution time ratios were computed as the mean values over the programs for which ratios are available, i.e. the first nine programs for CIFS/CIFI versus CSFS, and the first 11 programs for CIFI versus CIFS. The average on this test suite (see Table 5) says that CIFI analysis is 1 946 times faster than CSFS analysis, and 176 times faster than CIFS. The ratio between CIFS and CSFS times is, on average, 37.

Reaching definitions are usually computed with the purpose of determining the data dependencies in a program that are used in many applications, like, e.g. testing, reverse engineering and slicing.

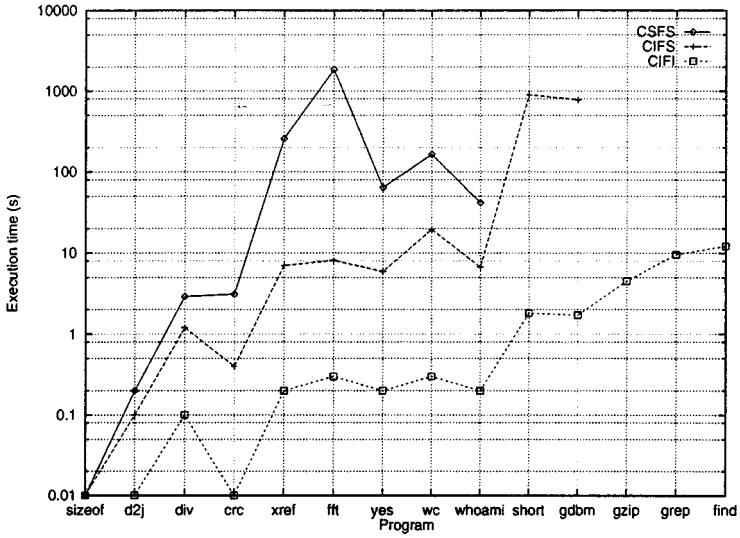

Figure 10. Computation times for the CSFS, CIFS and CIFI analyses in logarithmic scale

Table 5. Average execution time ratios: CIFS versus CSFS, CIFI versus CSFS and CIFI versus CIFS

	CIFS	CIFI
CSFS	37	1 946
CIFS		176

For this reason a comparison between the three proposed analyses is done with reference to data dependencies, instead of reaching definitions: the user of the results is usually interested in the data dependencies that can be extracted from reaching definitions. Data dependencies are obtained by considering the variables used in each program statement and extracting only those definitions that are relative to the used variables. The total number of data dependencies found out by an analysis variant depends on the amount of spurious definitions caused by flow and/or context insensitivity, and therefore can be used to compare the accuracy of the variants. Figure 11 shows the number of data dependencies detected by the three variants of the analyses in the programs of the test suite. The peak corresponding to the `fft` program is due to the kind of computation performed: `fft` computes the fast Fourier transform on any array of input data. A dynamic array is allocated for them, and is accessed by almost all the procedures that process data. Most data modifications refer to such a data structure, and result in the accumulation of a new data dependency. Therefore the array containing the processed data exhibits a high number of dependencies associated with every statement that modifies it. The high number of data dependencies in this program could also be an explanation of the results on execution times, where `fft` required a considerable amount of time relative to the other programs of comparable size. This issue will be discussed in detail later. Figure 12 depicts the percentage of data dependencies increase due to lower sensitivity to context/flow of CIFS and CIFI analyses. Spurious dependencies are added which could be eliminated by taking context/flow

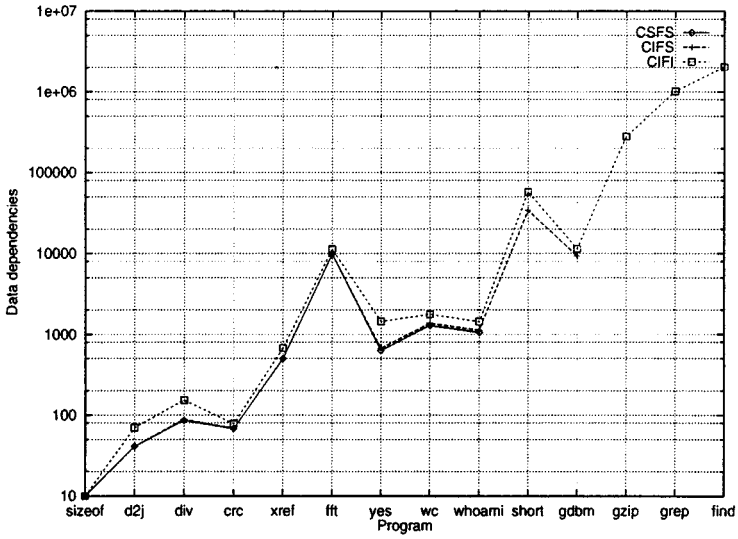

Figure 11. Data dependencies for the CSFS, CIFS and CIFI analyses in logarithmic scale

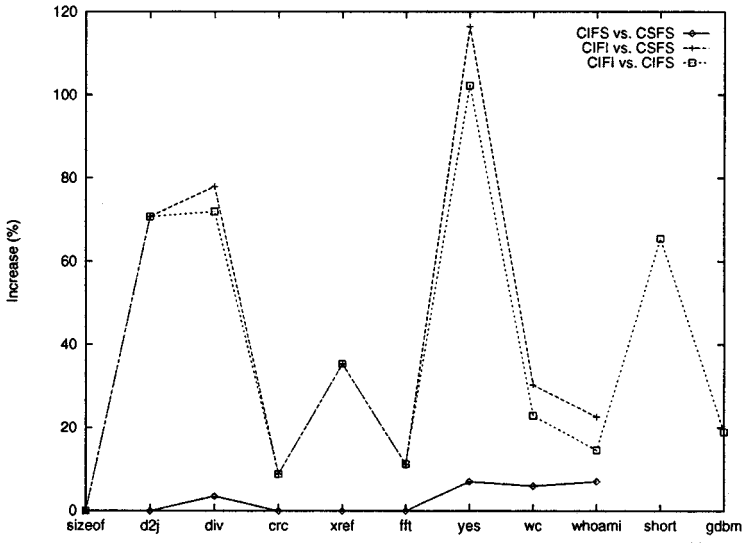

Figure 12. Data dependencies increase for the CSFS, CIFS and CIFI analyses

into consideration. The plots in Figure 12 show that the CIFI variant produces considerably more dependencies than both CIFS and CSFS. On the other hand, CIFS introduces a small number of spurious dependencies relative to CSFS, as apparent in the plot.

Table 6. Average data dependencies increases: CIFS versus CSFS, CIFI versus CSFS and CIFI versus CIFS

	CIFS	CIFI
CSFS	2%	41%
CIFS		38%

Average data dependencies increases were computed as the mean values over the programs for which increase data are available, i.e. the first nine programs for CIFS/CIFI versus CSFS, and the first 11 programs for CIFI versus CIFS (see Table 6). The CIFS data dependencies increase is on average 2%, while the CIFI is 38% and 41% versus CIFS and CSFS analyses, respectively. The program with the highest dependency increase is `yes`, while `fft`, which originates the highest number of data dependencies, exhibits quite small differences between CIFI and CIFS/CSFS variants. In fact, `fft` contains many operations on array components that are treated as non-killing definitions by our analysis, since the exact component that is defined is generally not known. As a consequence, more accurate analyses do not add precision, since they are unable to kill previous definitions when these are relative to array components. On the other hand, `yes` contains many definitions that may be labelled as killing if the control flow is considered. For this reason the CIFI variant introduces many spurious definitions, compared with CIFS/CSFS.

In the discussion of the complexity of the proposed algorithms, each elementary operation performed for each CFG node is assumed to have a constant cost. This is true if the cardinality of the reaching definitions set is small, so that the operations on it take a constant time. Such a hypothesis is generally satisfied by the programs in the test suite, with the exception of `fft`. In the `fft` program the high execution time may be caused by the high number of data dependencies to be discovered. To verify this hypothesis, the `fft` program was artificially modified by subtracting more and more variable definitions. To do that, the GEN and KILL sets of every node were reduced: a given percentage of definitions was randomly chosen to be subtracted from both of them. In this way 11 versions of `fft` with a decreasing number of variable definitions were generated. For each of them the CSFS analysis was performed and the resulting data dependencies were counted. Figure 13 shows the execution times on each of the 11 versions versus the number of data dependencies retrieved. It is clear that an increasing number of data dependencies produces an increase in execution times: programs with many dependencies are demanding in terms of execution time.

An additional test suite of medium-to-large size programs was considered to evaluate the scalability of the CIFI reaching definitions analysis. Table 7 contains three public domain and three industrial programs, with sizes evenly spread across a broad range, from 38 kLOC to 249 kLOC. Industrial program names are masked for confidentiality reasons. Their application domains span from telecommunications to computer-aided design and banking systems. Execution times remain below 30 seconds even on the two largest programs and have no dramatic increase with size. Therefore, the CIFI analysis is a feasible solution when the size of the analysed programs does not allow the use of more accurate algorithms. It can be noticed that the number of AST nodes is a better predictor of CIFI analysis performance than LOC, in that it is more correlated (0.98 versus 0.75) to the number of definitions that the analysis has to collect in linear time.

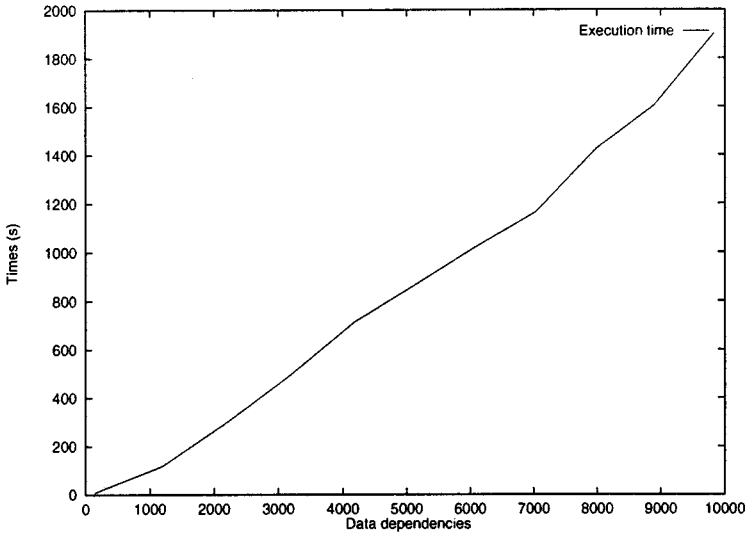

Figure 13. Execution times obtained for the `fft` *program by varying the number of data dependencies to be retrieved during the CSFS analysis*

Table 7. Reaching definitions computation times (s) for the CIFI analysis on three public domain and three industrial medium-to-large size programs

	Program	LOC	ULOC	AST nodes	Time (s)
1	`tar`	42 667	27 855	15 566	9.4
2	`vim`	43 014	30 740	29 556	18.2
3	`bash`	62 391	42 636	36 619	28.6
4	`ind-A`	38 319	24 705	4 710	2.2
5	`ind-B`	42 490	22 027	22 806	13.3
6	`ind-C`	249 711	245 703	18 858	20.4

To summarize, execution times exhibit a substantial decrease when the analysis becomes less sensitive to the calling context, and then to the control flow. Having a short response time can be extremely important when large systems are taken into account and the programmer needs quick responses. The precision loss associated with a lower sensitivity is small for CIFS analysis. In fact, it represents a very good trade-off between precision and computation time by giving high accuracy, on average 2% dependencies increase, with a considerable reduction in time, on average 37:1. The CIFI analysis offers a further speed-up of 1 946:1, but the price is an average 41% increase of the dependencies number. Actually, this analysis may still be useful when large industrial size systems have to be handled: when FLANT is used to search the statements defining a variable, the extra dependencies may increase the search space (and thus the search time), but the statement that is looked for is assured to be in the set of the dependencies. CIFI times remain quite low (20.4 s) even for the largest program (`ind-C`, 249 kLOC) in the industrial benchmark. In the authors' experience, larger industrial applications (>1 MLOC) are typically composed by distinct

communicating executables the size of which is usually comparable to `ind-C`'s, and therefore still analysable. A comparison of the computation times inside each analysis variant shows that they could be correlated also to the number of dependencies to discover, as in the case of the `fft` program, when this number is quite high.

4. CONCLUSION

Three variants of the interprocedural reaching definitions analysis have been presented in the context of software maintenance and understanding activities. The results have different degrees of accuracy, depending on the context and flow sensitivity, and require different levels of resources. FLANT, the analysis tool supporting the variable precision reaching definitions analyses, was conceived and developed to be modular and language independent. The results of the analyses can be in textual or graphical form, and can be displayed from a text editor, which allows interaction with the user from an editing environment.

The interprocedural reaching definitions analysis, in three versions, was performed on a test suite. The trade-off between precision and computation time was evaluated, and the results suggest that the maintainer should be enabled to choose between the variants, according to the analysis needs and the maintenance task being performed. A substantial execution time speed-up is obtained when passing from the CSFS to the CIFS and CIFI analyses: the average computation time for the whole test suite gains by a factor 37 and 176, respectively.

While the spurious dependencies introduced by the CIFS variant are a small fraction of the CSFS dependencies (2% on average), CIFI produces a considerable increase in number (41% on average). Furthermore, experimental results suggest that CIFI analysis can scale on large systems, since it was fast even on the largest program (249 kLOC) in the test suite, and its complexity is linear with the program size. Therefore, it offers a way to perform useful analyses even on large software systems, for which the other approaches are not feasible.

CSFS and CIFS analyses are capable of producing very precise results for programs of small-to-medium size. They fail to scale up to large systems, on which the CIFI variant can still complete the computation in a short time. The loss of precision is nevertheless not negligible. The choice among the three variants is influenced by the kind of application for which reaching definitions are computed.

The interactive access to data dependencies in a reverse engineering environment which allows one to modify and re-analyse the system requires a fast response time, and therefore CIFS may be a good answer for programs whose size is under 2 kLOC. On larger systems the user is likely to prefer the fast CIFI result with 41% spurious information, rather than remaining in wait for a long time interval.

On the other hand in applications like testing, for which reaching definitions are computed once for all to define the du-chains to be probed, the user could wait longer for the answer, since extra dependencies may have a very negative impact on successive steps. In such cases the CSFS is preferable, and should be replaced by CIFS and CIFI only when the more accurate variant is intolerably slow.

On-line applications like a smart code browser displaying approximate def-use information are possible users of the CIFI fast computation. On the other hand, architecture and design-concept

recovery tools are typically run off line, and therefore may take advantage of the most accurate results that can be computed at lower speed. On large systems the CIFI approximation is anyhow an acceptable substitute, as reported in Tonella *et al.* (1996).

Applications requiring the highest levels of accuracy even on large systems, like program transformation or testing, may take only partial advantage of the current proposed solutions. Thus, future work needs to be devoted to investigating how to increase the CIFI accuracy by combining its results with some degree of flow and context sensitivity.

APPENDIX A. ALGORITHM DESCRIPTION

A.1. CSFS algorithm

The CSFS algorithm performs a fix point over the functions in the call graph starting from `main`, as shown in Figure 14. The *Fixpoint* procedure in Figure 14 invokes the *Propagate* procedure repeatedly, until flow information does not change. The *Propagate* procedure takes into consideration all nodes in the analysed function, and propagates flow information inside each node, according to the rules presented above. When the considered node is a call node, a whole propagation has to be performed inside the called function, to be context sensitive. For this reason the incoming flow information is propagated at the entry of the called function and a fix point is run. When recursive calls are encountered, propagation must not be performed to avoid infinite loops. Recursive calls are dealt with by renouncing to distinguish between different recursive activations of the same function, and accumulating flow information for every recursive calling context. To recognize a recursive call, a trace of the call stack is kept by pushing each called function when encountering the respective call node, and popping it after the fix point has converged. If a call is found to be recursive, i.e. the called function is in the call stack, the incoming flow information is propagated at entry, as in normal cases, but no fix point is computed, so that the *Fixpoint* on such a function, already invoked, will be responsible for propagating the flow information from the recursive call together with its current flow information. The procedure *PropagateAtEntry* is responsible for transferring flow information both in the case of recursive and non-recursive calls. The transferred flow information is that related to global variables, invisible variables and formal parameters. The target language for the analyses is C, which only allows parameters to be passed *by value*: the call node defines the initial value of every formal parameter in the called function by means of the expressions supplied as actual parameters. Therefore, in the *PropagateAtEntry* procedure each formal parameter of the called function is considered as defined by the call node, through the corresponding actual parameter, and the reaching definition $\langle x, n \rangle$ is added, where x is the formal parameter and n is the call node.

The time complexity of this algorithm is $O(\exp(P)N^2)$, where P is the number of procedures and N is the maximum number of nodes per procedure in the program. In fact it is well known that intraprocedural flow analysis based on fix point has complexity $O(N^2)$. Such analysis is repeated for each calling context, i.e. $O(\exp(P))$ times in the worst case. If the assumption is made that `goto` instructions are never (or seldom) used in the program to perform backward jumps, the syntactic structure of the program can be exploited to obtain an efficient fix point, and consequently a more practical linear complexity can be derived in place of N^2. The exponential term is not


```
CSFSReachDef(CallGraph)
   Fixpoint(Main[CallGraph])

Fixpoint(Function)
   change ← true
   while(change)
       change ← Propagate(Function)

Propagate(Function)
   change ← false
   for each n in Function
       IN[n] ← ⋃_{p∈pred[n]} OUT[p]
       OLDOUT[n] ← OUT[n]
       if n is not a CALL NODE
          OUT[n] ← GEN[n] ⋃ (IN[n] - KILL[n])
       else
          PropagateAtEntry(CalledFunction[n], IN[n])
          if CalledFunction[n] is not in CallStack
              Push(CallStack, CalledFunction[n])
              Fixpoint(CalledFunction[n])
              Pop(CallStack, CalledFunction[n])
          OUT[n] ← GetReturnInfo(CalledFunction[n])
       if OUT[n] ≠ OLDOUT[n] then
           change ← true
   return change
```

Figure 14. Pseudo-code for the CSFS algorithm

only theoretically possible: according to the authors' experience it is the main cause for the limited scalability of this algorithm. Given a program consisting of L LOC (Lines Of Code), the time required to perform the CSFS analysis depends on the level of decomposition into procedures. At the two extremes the program may comprise one single procedure with L statements ($P = 1, N = L$), or L procedures with one statement each ($P = L, N = 1$) where each procedure calls only another procedure more than once. In the first case the quadratic term N^2 of the complexity is dominant, and N is the same order as the program size L, while in the second case the exponential term $\exp(P)$ becomes dominant, and P is the same order as the program size L. Therefore, in practice the performance of the analysis is somewhere between quadratic and exponential in the program size, depending on the level of decomposition into procedures and the organization of the procedures in the call graph. Here and in the following, the operations performed by each CFG node on the reaching definitions set are considered elementary, and are given a constant cost in the evaluation of the complexity. This is true if the involved sets of definitions have small cardinality, as it is reasonable to assume. Otherwise the number of definitions to be propagated into each node is another complexity factor to be introduced.


```
CIFSReachDef(CallGraph)
   Fixpoint(Main[CallGraph])

Fixpoint(Function)
    change ← true
    while(change)
        change ← Propagate(Function)

Propagate(Function)
   change ← false
   for each n in Function
       IN[n] ← ⋃_{p∈pred[n]} OUT[p]
       OLDOUT[n] ← OUT[n]
       if n is not a CALL NODE
             OUT[n] ← GEN[n] ⋃ (IN[n] - KILL[n]
       else
             AccumulateAtEntry(CalledFunction[n], IN[n])
             if IsLastCallingContext(n, CalledFunction[n])
                Fixpoint(CalledFunction[n])
              OUT[n] ← GetReturnInfo(CalledFunction[n])
       if OUT[n] ≠ OLDOUT[n] then
              change ← true
   return change
```

Figure 15. Pseudo-code for the CIFS algorithm

A.2. CIFS algorithm

The CIFS algorithm, shown in Figure 15, performs a fix point that repeatedly propagates flow information inside each function of the program, in a way similar to the CSFS analysis. The main difference is in how call nodes are dealt with. Being context insensitive, the propagation from a call node into the called function accumulates the incoming information, delays the actual propagation until the last calling context is encountered, and uses the cumulative flow information available at the exit of the called function. For this reason, for a call node, the *AccumulateAtEntry* is invoked to add current flow information at the entry of the called function. Then, the current calling context is tested to see if it is the last calling context, and only in such a case a fix point on the called function is computed. The procedure *IsLastCallingContext* uses the result of a preprocessing, not shown in Figure 15, necessary to mark the last calling context for every function, excluding recursive contexts. Such preprocessing is a simple call graph traversal in which the last visit of a call to a function is recorded, recursive calls excluded. The main drawback of this algorithm is that incomplete flow information is used by each call node that is not associated with the last calling context of the called function. The flow information that is inserted in the OUT set in such cases is retrieved by the *GetReturnInfo* procedure without performing a fix point on the called function, and therefore

it is the flow information currently available at the exit of the called function. On first iteration over the call graph such information is the empty set, while on successive iterations it is the flow information that was computed by the fix point of the previous iteration. Thus, it is not up to date with respect to modifications in the incoming flow information introduced by the current iteration. The *AccumulateAtEntry* procedure transfers flow information to the function called when referred to global variables, invisible variables and formal parameters, in a manner similar to the CSFS algorithm. Unlike the CSFS algorithm, where they are handled in an approximate context-insensitive way, recursive function calls do not need special treatment, because the same accumulation of non-recursive calls applies to them. The only problem may be in determining the last calling context: recursive calls must be excluded from the set of candidates to avoid the choice of a recursive call node, which is not reachable, since the propagation inside the enclosing function is delayed. A traversal of the interprocedural CFG, performed in the same order as in fix-point computation, is used to choose from the candidates the call node which is the last calling context for the called function.

The CIFS algorithm has a time complexity $O(P^2N^2)$, where P is the number of procedures in the program and N is the maximum number of nodes inside each procedure. In fact, the interprocedural CFG is treated as one single graph with $O(PN)$ nodes, on which a quadratic fix point is performed. The order in which subgraphs corresponding to procedures are considered during the fix point permits one to first accumulate the incoming flow information, and then propagate it inside the subgraph. This is the reason why propagation is delayed until the last calling context. The worst case complexity of the algorithm is the worst case complexity of a fix point over the whole interprocedural CFG, i.e. quadratic both in P and N. In practice the generally disciplined and limited use of `goto`s makes intraprocedural fix point performance closer to linear than quadratic. On the other hand, according to the authors' experience, this is not true for the term P^2.

A.3. CIFI algorithm

The CIFI algorithm, shown in Figure 16, first collects all variable definitions from each function in the program, and then propagates the collected definitions (the DEF set) inside every program function. The *CollectDef* procedure, used during the first step to collect the definitions of a function, simply computes the union of the GEN sets of all function nodes that are not call nodes. Call nodes need special treatment, as they implicitly introduce definitions of the formal parameters in the function called. It is assumed that the parameter passing mechanism is the *call by value*. Therefore, when a call node is encountered in the *CollectDef* procedure, each formal parameter of the called function is considered to be defined by the current node, and the reaching definition $\langle x, n\rangle$ is added to the current reaching definitions set, where x is the formal parameter and n is the call node. The *Propagate* procedure first determines the reaching definitions that are to be propagated inside the function. The criterion is to restrict the whole reaching definitions set to those visible from inside the function. This can be obtained by extracting the definitions that are either local to the function or global. Then, this set of definitions becomes the reaching definitions set of every node inside the given function. The context insensitivity of the CIFI algorithm does not require any special treatment for recursive function calls.

Since reaching definitions are collected without subtracting the KILL set, they result non-killing, consistent with the assumption of flow insensitivity. In fact, definitions killed along every possible


```
CIFIReachDef(CallGraph)
    DEF ← ∅
    for each f in CallGraph
        DEF ← DEF ⋃ CollectDef(f)
    for each f in CallGraph
        Propagate(f, DEF)

CollectDef(Function)
    SummaryDef ← ∅
    for each n in Function
        if n is not a CALL NODE
             SummaryDef ← SummaryDef ⋃ GEN[n]
        else
             for each x in FormalParameter[CalledFunction[n]]
                 SummaryDef ← SummaryDef ⋃{⟨ x, n ⟩}
    return SummaryDef

Propagate(Function, ReachDef)
    VisibleDef ← ∅
    for each x in Variable[ReachDef]
        if IsLocal(Function, x) or IsGlobal(x)
             VisibleDef ← VisibleDef ⋃ GetReachDef(x, ReachDef)
    for each n in Function
        IN[n] ← VisibleDef
        OUT[n] ← VisibleDef
```

Figure 16. Pseudo-code for the CIFI algorithm

path have to be retained in this analysis, since the control flow is ignored. In the second step, collected definitions are propagated inside every program function. The *Propagate* procedure adds a reaching definition to a function node if it is related to a variable local to the function or global. In this way, reaching definitions inside a function are inserted only for its local and global variables. Actually, the propagation step is not necessary, and one single node per function could be used as a representative of all nodes inside such a function, as the same flow information is otherwise replicated for every node. In this work the propagation step is assumed only to simplify the comparison with the other variants. As definitions of local and global variables are inserted together in the DEF set, it is a prerequisite that each variable has a unique identifier, in order to avoid clashing.

This algorithm considers in turn every function in the program, and for every function all nodes are considered inside the *CollectDef* and the *Propagate* procedures, respectively. For each function node a constant time operation is performed, so that the resulting worst case complexity is $O(PN)$, where N is the maximum number of nodes in the program functions and P is the total number of procedures.

References

Antoniol, G., Fiutem, R., Merlo, E. and Tonella, P. (1995) 'Application and user interface migration from basic to visual C++', in *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 76–85.

Atkinson, D. C. and Griswold, W. G. (1996) 'The design of whole-program analysis tools', in *Proceedings of the International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 16–27.

Callahan, D. (1988) 'The program summary graph and flow-sensitive interprocedural data flow analysis', in *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, ACM Press, New York, pp. 47–56.

Emami, M., Ghiya, R. and Hendren, L. J. (1994) 'Context-sensitive interprocedural points-to analysis in the presence of function pointers', in *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, ACM Press, New York, pp. 242–256.

Fiutem, R., Merlo, E., Antoniol, G. and Tonella, P. (1996a) 'Understanding the architecture of software systems', in *4th Workshop on Program Comprehension*, IEEE Computer Society Press, Los Alamitos CA, pp. 187–196.

Fiutem, R., Tonella, P., Antoniol, G. and Merlo, E. (1996b) 'A cliché based environment to support architectural reverse engineering', in *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 319–328.

Harris, D. R., Reubenstein, H. B. and Yeh, A. S. (1995) 'Reverse engineering to the architectural level', in *Proceedings of the International Conference on Software Engineering*, ACM Press, New York, pp. 186–195.

Horwitz, S., Pfeiffer, P. and Reps, T. (1989) 'Dependence analysis for pointer variables', *SIGPLAN Notices*, **24**(7), 28–40.

Horwitz, S., Reps, T. and Binkley, D. (1988) 'Interprocedural slicing using dependence graphs', in *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, ACM Press, New York, pp. 35–46.

Kozaczynski, V., Ning, J. Q. and Engberts, A. (1992) 'Program concept recognition and transformation', *IEEE Transactions on Software Engineering*, **18**(12), 1065–1075.

Landi, W. and Ryder, B. G. (1992) 'A safe approximate algorithm for interprocedural pointer aliasing', in *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, ACM Press, New York, pp. 235–248.

Pande, H. D., Landi, W. A. and Ryder, B. G. (1994) 'Interprocedural def-use associations for C systems with single level pointers', *IEEE Transactions on Software Engineering*, **20**(5), 385–403.

Steensgaard, B. (1996a) 'Points-to analysis in almost linear time', in *Proceedings of the 23rd AGM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, New York, pp. 32–41.

Steensgaard, B. (1996b) 'Points-to analysis by type inference of progams with structures and unions', in *Proceedings of the International Conference on Compiler Construction*, no. 1060 in *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 136–150.

Stocks, P. A., Ryder, B. G., Landi, W. A. and Zhang, S. (1998) 'Comparing flow and context sensitivity on the modification-side-effects problem', in *Proceedings of the International Symposium on Software Testing and Analysis*, ACM Press, New York, pp. 21–31.

Tonella, P., Antoniol, G., Fiutem, R. and Merlo, E. (1997) 'Flow insensitive C++ pointers and polymorphism analysis and its application to slicing', in *Proceedings of the International Conference on Software Engineering*, ACM Press, New York, pp. 433–443.

Tonella, P., Fiutem, R., Antoniol, G. and Merlo, E. (1996) 'Augmenting pattern-based architectural recovery with flow analysis: mosaic—a case study', in *Proceedings of the Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 198–207.

Weiser, M. (1984) 'Program slicing', *IEEE Transactions on Software Engineering*, **10**(4), 352–357.

Wilson, R. P. and Lam, M. S. (1995) 'Efficient context-sensitive pointer analysis for C programs', in *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, ACM Press, New York, pp. 1–12.

Author's biography:

Ettore Merlo is an Associate Professor of Computer Science at the Ecole Polytechnique de Montreal. Previously he was the lead researcher of the software engineering group at CRIM (Computer Research Institute of Montreal). His research interests are in software analysis, software re-engineering, user interfaces, software maintenance, and artificial intelligence. He has collaborated with several organizations on projects in software re-engineering, clone detection, software quality assessment, and architectural reverse engineering. Ettore's Ph.D. is in Computer Science from McGill University in Montreal, and his Laurea degree summa cum laude is from the University of Turin in Italy. His email address is ettore.merlo@polymtl.ca